

Advances in the Planarization Method: Effective Multiple Edge Insertions

*Markus Chimani*¹ *Carsten Gutwenger*²

¹Institute of Computer Science, FSU Jena

²Department of Computer Science, TU Dortmund

Abstract

The planarization method is the strongest known method to heuristically find good solutions to the general crossing number problem in graphs: Starting from a planar subgraph, one iteratively inserts edges, representing crossings via dummy vertices.

In the recent years, several improvements both from the practical and the theoretical point of view have been made. We review these advances and conduct an extensive study of the algorithms' practical implications. Thereby, we also present the first implementation of an approximation algorithm for the crossing number problem of general graphs. We compare the obtained results with known exact crossing number solutions and show that modern techniques allow surprisingly tight results in practice.

| | | | |
|--------------------------------|-----------|------------------|-----------|
| Submitted: | Reviewed: | Revised: | Accepted: |
| | Final: | Published: | |
| Article type: Regular Paper | | Communicated by: | |

The research was partially supported by EUROGIGA project GraDR 10-EuroGIGA-OP-003.

Markus Chimani was funded by a Carl-Zeiss-Foundation juniorprofessorship.

E-mail addresses: markus.chimani@uni-jena.de (Markus Chimani) carsten.gutwenger@cs.tu-dortmund.de (Carsten Gutwenger)

1 Introduction

Given a graph $G = (V, E)$, the *crossing number* problem asks how to draw G into the plane with the fewest possible number of edge crossings. The *planarization method* is the probably best known and most successful heuristic to tackle the crossing number problem in practice. In its simplest form it runs in two phases: first, a (large) planar subgraph $G' = (V, E')$ of G is computed. Then, the temporarily removed edges $F := E \setminus E'$ are re-inserted one after another, each time solving a *single edge insertion* problem. This problem can be stated as follows: Let H be a planar graph and e an edge not yet in H . We search for a smallest planar graph H^+ which represents a drawing of $H + e$ where edge crossings are replaced by dummy vertices of degree 4, and all these crossings occur on the edge e . Hence, when removing the image of e from H^+ , we obtain a planar embedded graph H . Using this method, each edge of F is inserted in a planar graph until we obtain a *planarization* G^+ , representing G in a planar way by using dummy vertices for crossings.

In the first proposal [2] of this heuristic, the insertion problem was considered with respect to a *fixed* embedding (cyclic order of the edges around their incident vertices) of the planar graph H . (I.e., after obtaining the planar subgraph G' , one embedding of G' is fixed and retained throughout the whole insertion phase.) A simple linear-time BFS-algorithm in the dual graph of H suffices to find an optimal solution. Later, and rather surprisingly, it was shown in [18] that there exists a linear-time algorithm, using the SPQR-tree data structure, which finds the optimal insertion path for e over all possible planar embeddings of H . In [17] it was shown that this approach is in practice vastly superior to the former in terms of the overall obtained number of crossings.

In recent years it was furthermore shown that there exists a (rather complex) insertion algorithm [6] to optimally insert a vertex with all its incident edges into a planar graph H , considering all possible embeddings of H (*vertex insertion*). On the other hand, it is NP-hard to insert an arbitrary set of edges simultaneously (*multiple edge insertion*), even when the embedding of the planar graph into which to insert is fixed [35].

Most interestingly, the single edge insertion problem (over all possible embeddings of H) is known to approximate the crossing number of $H + e$ within a factor of $\Delta/2$, where Δ is the graph's maximum (vertex-)degree [4, 19]. I.e., there exists a drawing of $H + e$ in which no two edges of H cross and whose number of crossings is at most $\Delta/2$ times the crossing number of $H + e$. Similarly, the vertex insertion problem approximates the crossing number of the resulting graph [8]. In particular, the proof of the latter can be generalized to show that an optimal multiple edge insertion solution—with respect to an edge set F and over all possible embeddings of G' —approximates the crossing number of $G' + F$ within a factor only dependent on Δ and $|F|$ [8].

Hence, the question arose whether this multiple edge insertion problem can be efficiently approximated. After a rather complicated approach in [10], a simpler and at the same time approximation-wise stronger algorithm was presented only recently [7]. The algorithm reuses concepts of the SPQR-tree based single

edge insertion and seems simple enough to be implemented and used in practice. The latter paper also shows that the traditional iterative single edge insertion algorithm cannot be an approximation strategy for the crossing number of G .

Contribution. In this paper we present recent advances of the planarization approach from a practical point of view. On the one hand, we show how to improve on the traditional approach of iteratively inserting single edges, via the use of strong postprocessing routines. On the other hand, we give the first implementation of a simultaneous multiple edge insertion algorithm—hence, this is also the first practical study of any crossing number approximation algorithm for arbitrary graphs. By considering graph classes of known crossing numbers (either from theory or from the application of the currently strongest branch-and-cut based exact crossing minimization algorithm [9]) we can deduce a practically very good performance of these heuristics: they usually find optimum, or at least very-close-to-optimum, solutions.

In the next section we recapitulate the central decomposition data structures used for the insertion algorithms. We then summarize the general strategies and central steps of these algorithms chiefly in Section 3. Based thereon, Section 4 describes improvements and implementation and engineering choices, allowing the algorithms to work well in practice. Finally, Section 5 contains the full discussion of our experimental evaluation, and Section 6 concludes the paper with some interesting open problems.

2 Preliminaries

In order to present our algorithmic choices and modifications, we first have to briefly introduce two central decomposition structures, used in all algorithms dealing with the insertion problem over all possible embeddings of H .

Let H be a connected graph. The *BC-tree* $\mathcal{B} = \mathcal{B}(H)$ of H is a tree with two different node types B and C: For each cut vertex in H , \mathcal{B} contains a unique corresponding C-node, and for each *block*, i.e., a maximal two-connected subgraph or a bridge, in H a unique corresponding B-node. Two nodes in \mathcal{B} are adjacent if and only if they correspond to a block b and a cut vertex c , such that $c \in b$. It is well-known that the size of \mathcal{B} is linear in the size of H , and that \mathcal{B} can be constructed in linear time, by computing the biconnected components of H ; see [21, 34].

Based thereon, we can further decompose each non-trivial block H' (i.e., a block that is not a bridge) via an SPQR-tree [12, 13] into its triconnected components: While SPQR-trees are more complicated than BC-trees, they also only require linear size and can be constructed in linear time [16, 20]. This data structure is particularly interesting, as it directly encodes all (exponentially many) planar embeddings of H' . We use the definition from [5, 7] which does not use Q-nodes, and therefore call the decomposition *SPR-tree* for conciseness. Chiefly summarizing, each tree node corresponds to a *skeleton*, a “sketch” of H' where certain subgraphs are replaced by virtual edges; a skeleton’s structure is

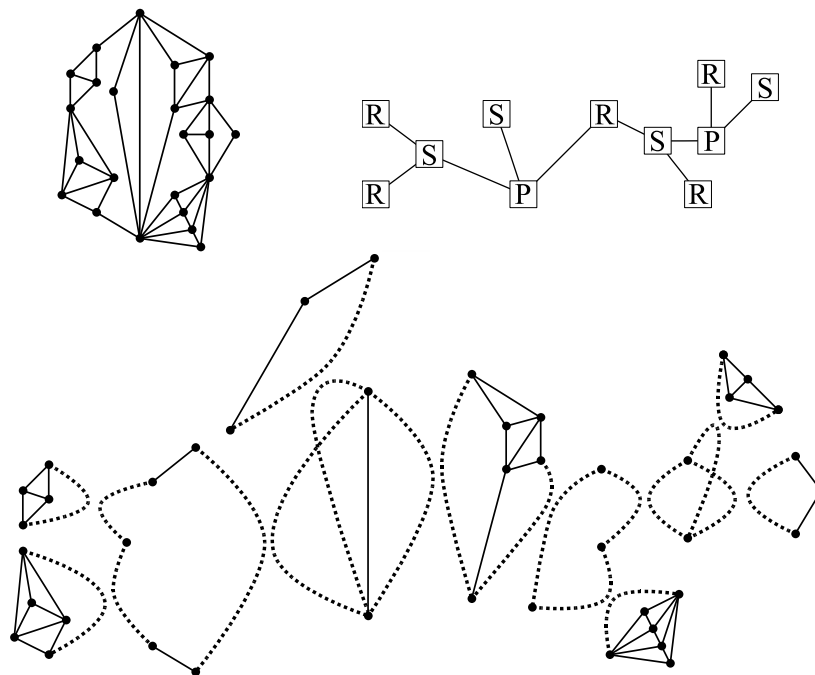


Figure 1: A planar graph (top left), its SPR-tree (top right), and its decomposition showing the skeletons (bottom). Virtual edges are represented by dotted lines.

restricted to only three simple types. By repeatedly merging the skeletons of adjacent nodes (at their virtual edges representing each other), we can obtain the original graph. See Figure 1 for an example.

Definition 1 (SPR-tree). Let H' be a biconnected graph with at least three vertices. The *SPR-tree* $\mathcal{T} = \mathcal{T}(H')$ of H' is the (unique) smallest tree satisfying the following properties:

- i. Each node ν in \mathcal{T} corresponds to a *skeleton* $S_\nu = (V_\nu, E_\nu)$ which is a “sketch” (minor, in fact) of H : Certain subgraphs are replaced by single *virtual edges*. The non-virtual edges are referred to as *original edges*.
- ii. The tree has three different node types with specific skeleton structure:
 - S:** The skeleton is a simple cycle; it represents a *serial* component.
 - P:** The skeleton consists of two vertices and multiple edges between them; it represents a *parallel* component.
 - R:** The skeleton is a simple triconnected graph. Note that a planar triconnected graph has a unique embedding (up to mirroring).
- iii. For the edge (ν, μ) in \mathcal{T} we have: S_ν contains a virtual edge e_μ which represents the subgraph described by S_μ , and vice versa.

- iv. We can obtain the original graph H' by iteratively merging the skeletons of adjacent tree nodes: For the edge (ν, μ) in \mathcal{T} , let e_μ (e_ν) be the virtual edge in ν (μ) representing the subgraph described by S_μ (S_ν , respectively). Clearly, both edges e_μ and e_ν connect the same nodes, say u and v . We obtain a merged graph $(V_\nu \cup V_\mu, E_\nu \cup E_\mu \setminus (e_\mu, e_\nu))$ by gluing the graph together at u and v and removing e_μ and e_ν .

Observe that the merge operation guarantees that the end vertices of a virtual edge are in fact a 2-cut (a *split pair*), i.e., their removal splits the graph in two or more components. In fact, the skeletons are exactly the triconnected components of H' discussed in [20].

In the algorithmic description of the multiple edge insertion approximation algorithm [7], an amalgamated version of these trees is considered:

Definition 2 (Con-tree). Let H be a connected planar graph. The *con-tree* $\mathcal{C} = \mathcal{C}(H)$ is formed by the BC-tree $\mathcal{B}(H)$ which holds SPR-trees $\mathcal{T}(H')$ for all non-trivial blocks H' of H .

Clearly, the linear-sized con-tree \mathcal{C} can be obtained from H in linear time.

3 Planarization Approach

In the above sketched planarization scheme, we can assume that the original graph G is connected, as otherwise the crossing number problem decomposes into multiple independent problems. Furthermore, the initial planar subgraph G' can be assumed to be maximal (i.e., no edge of G can be added without losing planarity) and hence also connected. For the single edge insertion algorithms, we will usually consider any intermediate graph H ; for the multiple edge insertion algorithm we set $H := G'$.

3.1 Single Edge Insertion

We will briefly recapitulate the central ingredients of the exact linear-time algorithm by Gutwenger et al. [18] to solve the single edge insertion problem over all possible embeddings of H . Let v_1, v_2 be the vertices we want to connect in H via a new edge. First consider a fixed embedding of H and let H_D be its dual. We define an *insertion path* to be a path in H_D connecting a face incident to v_1 with a face incident to v_2 . The length of this path is then the number of edge crossings necessary to insert the edge $\{v_1, v_2\}$ into embedded H along this path; each dual edge in the insertion path corresponds to an edge in H that is to be crossed. We can directly compute the shortest insertion path via standard breadth-first search (BFS).

Now consider H with variable embedding. Let L be the unique shortest path in $\mathcal{B}(H)$ from a B-node containing v_1 to a B-node containing v_2 . The optimal

insertion path for $\{v_1, v_2\}$ in G can be obtained by concatenating the optimal insertion paths within the (non-trivial) blocks on this path L ; we can always nest blocks at a common cut vertex into each other such that there arise no additional crossings. For a block H' represented by a B-node on L , let $v_i^{H'}$, $i = 1, 2$, denote v_i if $v_i \in V(H')$, or the cut vertex in H' closest to v_i otherwise. It remains to find optimal insertion paths from any face incident to $v_1^{H'}$ to any face incident to $v_2^{H'}$, for each non-trivial block H' .

Therefore, let $Q_{H'}$ be the unique shortest path in $\mathcal{T}(H')$ from a skeleton containing $v_1^{H'}$ to a skeleton containing $v_2^{H'}$. It was shown in [18] that only the embeddings of the skeletons along $Q_{H'}$ matter. In a nutshell, the algorithm walks along these skeletons and fixes suitable embeddings for the skeletons, one after another. Finally, an optimal embedding is found and fixed, and one can use the simple BFS algorithm on the dual graph to insert the edge $\{v_1^{H'}, v_2^{H'}\}$ optimally.

In the following, we can consider a *con-chain* Q in $\mathcal{C}(H)$ of the edge $\{v_1, v_2\}$ as an extended version of L , where the “subpaths” $Q_{H'}$ are stored at each non-trivial block H' along L .

3.2 Multiple Edge Insertion

Let us briefly review the approximation algorithm for the multiple edge insertion problem by Chimani and Hliněný [7]. Let $H := G'$ be the initial planar subgraph of G into which to insert the edges in $F = \{e_i\}_{1 \leq i \leq |F|}$. Assume we could independently insert each edge $e_i \in F$ into H . Using the above algorithm for single edge insertions, we would obtain a con-chain Q_i for each edge e_i , and therefore a so-called *embedding preference* for each node on Q_i with respect to e_i . We obtain a common embedding of H via an iterative voting scheme on the (in general conflicting) embedding preferences per con-tree node; see also Section 4.2. Among other properties, the scheme ensures that at any con-tree node at least one preference is satisfied. After realizing the so-chosen embedding, we can once again use the simple BFS algorithm in the dual graph to insert all edges of F (iteratively) into this fixed embedding.

The prove-wise crucial part in the algorithm is that any two con-chains Q_i, Q_j are either disjoint or they intersect in one sub-chain. Hence, two con-chains (think of simple paths) “deviate” at at most two nodes in the con-tree (think of a usual tree): once when the two paths come together and once when they part. It is shown in [7] that the embedding preferences may conflict only very locally at these two “places” (called *passes*). Thereby, the exact definition of *pass* is quite involved; here, it suffices to state that such a pass might span over at most five con-tree nodes. Overall we can bound the number of nodes where some con-chains disagree on the embedding preference, as well as the additionally necessary number of crossings to route an edge through a skeleton that is differently embedded than desired. This gives an approximation factor for the optimal multiple edge insertion with respect to G' and F , and, subsequently, for the crossing number of G .

It remains to clarify what an *embedding preference* actually is: Observe that

S-nodes do not allow different embeddings of their skeletons. For an R-node (a triconnected planar graph), we have only a unique planar embedding and its mirror. For a P-node, each inserted edge may want two particular edges of the skeleton to be cyclicly adjacent (in, say, clockwise direction). Finally, for a C-node each inserted edge may want a particular incident face in an adjacent block to be identified with a particular incident face in another adjacent block.

4 Engineering

4.1 Iterative Single Edge Insertion and Postprocessing

In the traditional planarization heuristic, we will “simply” insert the temporarily removed edges F one after another into the planar subgraph. After each insertion, we replace the arising crossings by dummy vertices, and hence proceed with a planar graph. There are various ways to fine-tune the obtained result via postprocessing, as already discussed in [17]. The simplest—and in fact quite effective—variant is to start the insertion process multiple times, each time with a different, randomized order of F . Additionally, each such insertion run can be improved: After having inserted all edges, we can again remove some original edge e from the planarization (i.e., we remove all the subedges and dummy vertices that represent e), and re-insert it, possibly requiring fewer crossings. For this operation, we can consider either the inserted edges F (*ins*), all edges (*all*), or the $x\%$ of the edges with the most crossings (*most x* , for some constant x). In [17] it was shown, that these approaches lead to greatly improved results.

Herein, we propose a further improvement on these methods. The *incremental (inc)* strategy basically applies the *all* strategy after each single insertion step. I.e., after the insertion of an edge $e \in F$, we try to remove and reinsert every other edge already in the graph, in order to obtain a better crossing number, before proceeding with the next edge from F . We will see that this approach again dominates the previously best strategy *all*, though at the cost of a vastly increased running time.

Note that all these strategies—when applied in a fixed embedding setting—are also applicable to the multi-edge insertion problem, after fixing an embedding into which all edges in F need to be inserted. Formally, the *inc* setting has to restrict itself to only try to reinsert the edges in F , in order to retain the approximation guarantee. Interestingly, after having obtained a postprocessed solution in the fixed embedding, we can run the *all* postprocessing where the graph’s embedding may change, i.e., using the optimal edge insertion over all possible embeddings! As the solution value never increases, the algorithm retains its approximation guarantee and improves the number of crossings in practice.

From the approximation point of view, we can observe that the first part of the algorithm (fixing a suitable overall embedding) tries to minimize the number of crossings between edges in F and edges in G' , while the postprocessing routines most importantly try to reduce the number of crossings between edges in

F —their quantity can only be estimated as $\binom{|F|}{2}$ in the formal quality guarantee.

4.2 Implementing Multiple Edge Insertion

In [7], certain aspects of the multiple edge insertion algorithm are described to be suitable for a comparatively smooth approximation proof. When implementing the algorithm, we take some different, though completely equivalent, routes. A main point of deviation is the consideration of *dirty passes*. We highlight the two main divergent choices here. Overall, our viewpoint allows a quite simpler implementation than would be easily deduced from the theoretical proofs of [7] alone.

The formal definition of dirty passes is very technical and lengthy, and requires a couple of supplementary definitions. Hence we refer the reader to [7] for the details, and we will only give a rough overview herein, transporting the idea but glossing over several details. Conceptually, a dirty pass describes a “place” in the con-tree where multiple insertion paths disagree on their preferred embedding. Roughly speaking, this can only happen whenever two insertion paths meet in the con-tree, coming from different branches of the tree, cf. Figure 2. When satisfying the preference for one of the insertion paths, the others may have to be routed through a “wrongly embedded” subgraph: such a routing requires up to $\Delta/2$ additional crossings. Hence, as an overall goal, the algorithm has to make embedding decisions in order to minimize the number of dirty passes. The “place” such a dirty pass describes is a subpath of 1–5 con-tree nodes (i.e., 1–3 non-S-nodes, possibly interleaved by S-nodes); hence, based on the node’s types and embeddings, a routing through several unfortunate embeddings may constitute a single dirty pass, or multiple distinct dirty passes. Figure 3 shows two simple cases of dirty passes, consisting of a single P-node and of two adjacent R-nodes, respectively.

In [7] a sizable number of different dirty pass types, including a tie-breaking scheme to prohibit invalid overlaps of dirty passes, is required. Yet, from the proofs it becomes clear that this is merely necessary to correctly estimate the *number* of these passes. Within the algorithm, these passes are only detected in order to identify possible flips to prohibit too many such situations. In other words: many dirty passes are unavoidable anyhow; the algorithm can only try to reduce a subclass of all possible dirty passes, namely, only dirty passes where a flip along such a pass improves the situation locally.

Con-Tree. Originally, the con-tree as an amalgamated version of BC- and SPR-trees is proposed, which allows to talk about a single con-chain (path) for each inserted edge. In the implementation, we perform the algorithm differently: First, we compute a suitable combinatorial embedding for each non-trivial block independently. Only then, we consider the C-nodes at which the blocks are joined. From the formal definition of dirty passes, we can easily deduce that C-nodes do not interact with other nodes in terms of realized embedding preferences, and hence we can independently choose which faces to embed into each other at cut vertices, after fixing the embeddings of the incident blocks.

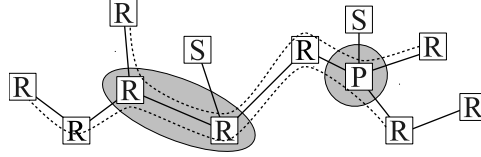
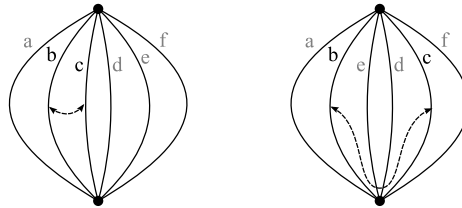
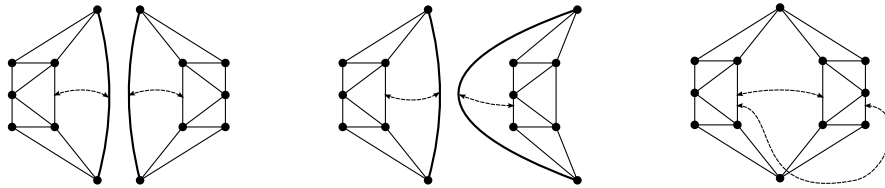


Figure 2: How dirty passes arise: the dotted lines denote insertion paths of two different edges within an SPR-tree: when two insertion paths meet, their embedding preferences may disagree—not only at a single node, but at some subpath in the tree. The gray regions denote possible dirty passes.



(a) An insertion path through a P-node requires no crossings for single edge insertion, as we can find a permutation such that the two “important” edges are next to each other. When fixing a different (“wrong”) embedding, the insertion path may cross over roughly half of the edges incident to one of the skeleton’s two vertices.



(b) Two adjacent R-nodes; the thick edges denote the virtual edges corresponding to this adjacency (i.e., each thick edge represents the SPR-subtree rooted at the other R-node). Recall that R-nodes allow only a unique embedding and its mirror. Consider two different insertion paths traversing both R-nodes. While one path (left) prefers both nodes in their “default” embedding, the other path (middle) would want one of the R-nodes mirrored. Not mirroring any R-node leads to additional (up to $\Delta/2$) crossings for the second path (right).

Figure 3: Two example of dirty passes. Insertion paths are denoted by dashed edges; arrows pointing at a (virtual) edge of a skeleton means that an insertion path will enter the component represented by this virtual edge. In the examples, all edges can be thought of as being virtual edges representing thick subgraphs.

This modification allows us to consider only two-connected graphs and SPR-trees in the following, vastly simplifying implementation details as most of the software infrastructure necessary for single edge insertions can be reused.

Merging the Embedding & Repairing Dirty Passes. As sketched above, the algorithm only has to identify possible flips to prohibit too many dirty passes. For this purpose alone, a much simpler strategy than a full-blown dirty pass classification suffices: Usually, we consider one insertion path after another. We traverse its SPR-nodes and fix the embedding of each skeleton along this path as preferred. When a visited node already has a fixed embedding, we try to flip the predecessor nodes of our current path in order to avoid dirty passes. Instead of checking the full case distinction in the dirty pass definition, it suffices to consider the case where the currently visited nodes ν and its predecessor (disregarding S-nodes) μ are P- and/or R-nodes:

We say that an embedding preference at a P-node *agrees* with a fixed embedding of this node’s skeleton, if the specified two edges occur clockwise neighboringly. An embedding preference of an R-node is simply a binary flag specifying whether to use a “default” planar embedding of the node’s skeleton or the “mirror” (only these two embeddings exist). Now, we only have to flip μ and its predecessors¹ along the insertion path iff μ and ν are *switching*, i.e., the new embedding preferences agree with the already fixed embedding of one of these two nodes, and agrees with the *flipped* embedding of the other node.

Doing this for all such pairs ν, μ then also repairs dirty passes on node triples, if at all possible. In all other cases of dirty passes, no flip can improve the situation anyway and hence is not necessary. It is understood that this procedure performs the same flips as the more abstract merge routine described in [7], and hence the implementation retains the approximation guarantee.

5 Experiments

5.1 Experimental Setup

We implemented all algorithms using the C++ library OGDF² and ran our experiments on a Linux system with an Intel Core i7-940 processor (2.93 GHz, 8MB cache) and 12 GB RAM. For each graph instance, all edge insertion algorithms were called with the same, pre-computed maximal planar subgraph, which was computed using OGDF’s PQ-tree based planar subgraph algorithm [22] (best of 250 random runs, i.e., random choices of the initial *st*-edge for the *st*-numbering) and iteratively adding removed edges afterwards if they do not destroy planarity (hence, all planar subgraphs were maximal).

¹As in the original algorithm, we then also have to flip further nodes whose embedding is already fixed and who are adjacent to flipped nodes. But herein, this can be viewed as only a minor technicality.

²Open Graph Drawing Framework, see <http://www.ogdf.net>.

5.2 Benchmark Sets

For creating our benchmark sets, we applied a reduction strategy to the graphs, which removes parts of the graphs that are irrelevant for planarization-based crossing minimization heuristics: self-loops, parallel edges, and planar blocks. Furthermore, we iteratively remove each vertex v with degree two not contained in a 3-cycle and add an edge connecting its two neighbors (the condition that v is not part of a 3-cycle prevents us from introducing parallel edges). Due to the removal of planar biconnected components, the resulting graphs can be disconnected; in such a case, we considered each connected component separately as a graph.

We created four benchmark sets, namely the *Rome*, *AT&T*, *ISCA*, and *KnownCR* graphs, as described in the following. Our whole benchmark set (including the planar subgraphs we used) can be downloaded at:

<http://ls11-www.cs.uni-dortmund.de/people/gutweng/planexp.zip>

Rome Graphs. This benchmark set is based on the well-known *Rome library* [11], a collection of 11,528 planar and non-planar graphs ranging from 10 to 100 vertices. These graphs are quite sparse with an average density of 1.35. We applied our reduction strategy to all graphs and selected the resulting non-planar graphs with at least 25 vertices and at least two edges removed in the computed planar subgraphs, which are 1843 graphs with 25–58 vertices³.

Figure 4 gives an overview on the Rome benchmark set, displaying the number of graphs and the average number of edges per vertex count. Furthermore, it shows the average number of edges deleted in the planar subgraphs and for how many of the graphs we know the exact crossing number from the branch-and-cut algorithm presented in [5, 9].

AT&T Graphs. The *AT&T* graphs are another well-known benchmark set (available at <http://graphdrawing.org/data.html>), containing directed and undirected graphs. We considered all graphs as undirected and applied the same reduction strategy and selection criteria as for the Rome graphs. The resulting benchmark set consists of 311 graphs with 25–312 vertices.

Figure 5 gives an overview on the AT&T benchmark set. The graphs are grouped according to the number of crossings in the best found solutions (this is the same grouping as used in Figure 12 and 13; see evaluation below). The diagram shows the number of graphs in each group (left vertical axis), as well as the average number of vertices, edges, and edges deleted in the planar subgraphs (right vertical axis).

³These lower bounds prohibit graphs with too trivial structures with respect to the number of P- and R-nodes, as well as the number of edges removed in the planar subgraph. Furthermore the “25” is chosen so that we have a significant number of graphs for each vertex count (at least for the smaller graphs).

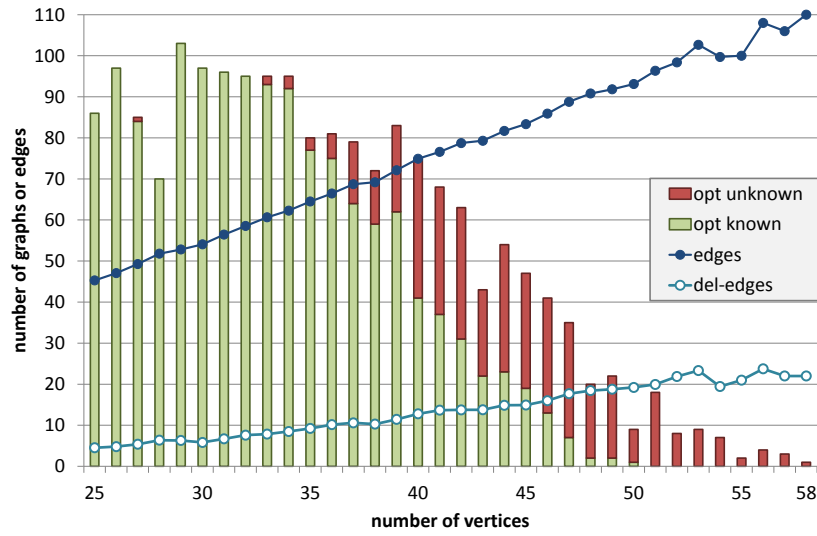


Figure 4: Statistics on the *Rome* graphs.

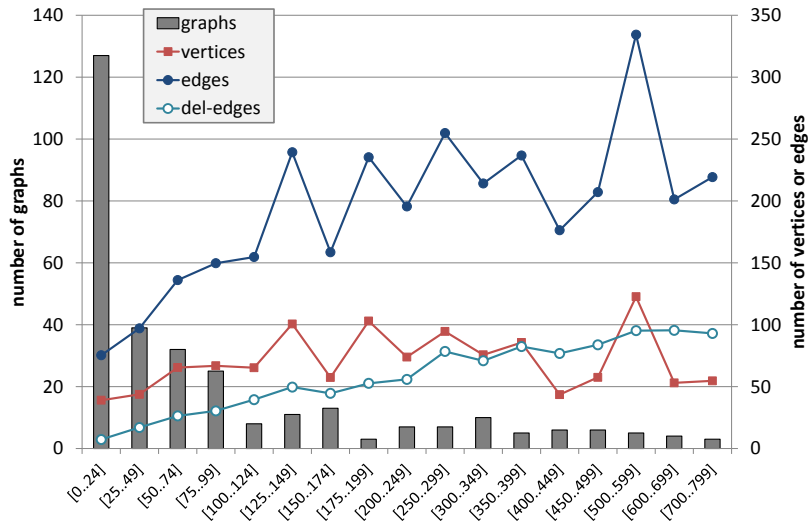


Figure 5: Statistics on the *AT&T* graphs.

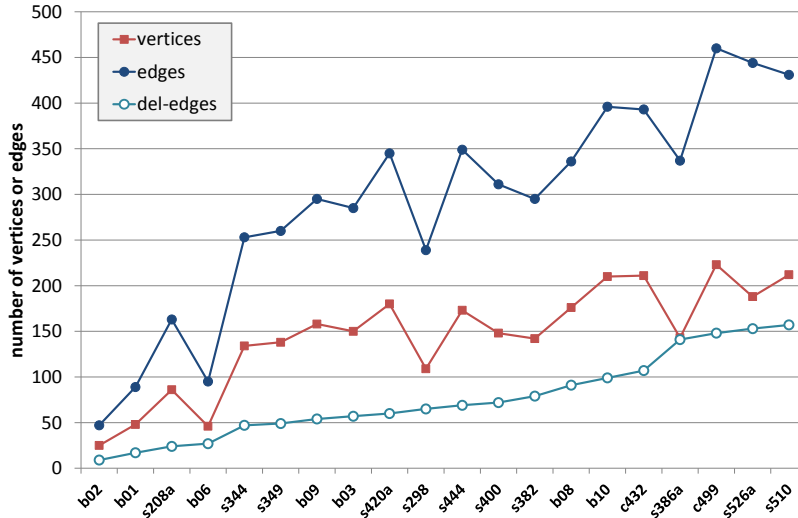


Figure 6: Statistics on the *ISCA* graphs.

ISCA Graphs. The *ISCA* graphs are hypergraphs taken from the *ISCA*'85 benchmark set of real world electrical networks, transformed into traditional graphs by substituting each hyperedge h by a new hypervertex connected to all vertices contained in h , connecting all inputs (outputs) to a new vertex s_{in} (s_{out} , resp.), and introducing the edge (s_{in}, s_{out}) . We used the same reduction and selection strategies as described above leading to 20 graphs with 25–223 vertices.

Figure 6 gives an overview on the *ISCA* benchmark set; for each graph the diagram shows its number of vertices, number of edges, and number of edges deleted in the planar subgraph.

KnownCR Graphs. Finally, the *KnownCR* graphs have been introduced in [15]. They are a collection of 1946 graphs with 9–250 vertices, for which the crossing numbers are known by theoretical proofs. Let C_n denote the cycle with n edges and P_n the path with n edges. Recall that, given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, their Cartesian product $G_1 \square G_2$ is the graph that has all possible vertex pairs $V_1 \times V_2$ as its vertices and holds the edges $\{(v_1 u_2, w_1 u_2) \mid (v_1, w_1) \in E_1, u_2 \in V_2\} \cup \{(u_1 v_2, u_1 w_2) \mid u_1 \in V_1, (v_2, w_2) \in E_2\}$.

The benchmark set contains four classes of graphs:

- For Cartesian products of cycles $C_m \square C_n$ with $3 \leq m \leq 7$ and $n \geq m$, the crossing number is $n(m-2)$ [1]. The collection contains 251 of these graphs with $nm \leq 250$.
- Various results have been published for the crossing numbers of Cartesian products of 5-vertex graphs G_i with paths P_n ; see Table 1 in the Appendix.

The collection contains 893 such graphs with $3 \leq n \leq 49$.

- Similarly, there are several theoretical results for the crossing numbers of Cartesian products of 5-vertex graphs G_i with cycles C_n ; see Table 2 in the Appendix. The collection contains 624 such graphs with $3 \leq n \leq 50$.
- Finally, the collection contains generalized Petersen graphs $P(m, 2)$ and $P(m, 3)$ ⁴. Exo et al. [14] have shown that $P(2k, 2)$ is planar and

$$\begin{aligned} \text{cr}(P(5, 2)) &= 2 \\ \text{cr}(P(2k + 1, 2)) &= 3 \quad \text{for } k \geq 3. \end{aligned}$$

Richter and Salazar [32] have shown that

$$\begin{aligned} \text{cr}(P(9, 3)) &= 2 \\ \text{cr}(P(3k, 3)) &= k \quad \text{for } k \geq 4 \\ \text{cr}(P(3k + 1, 3)) &= k + 3 \quad \text{for } k \geq 3 \\ \text{cr}(P(3k + 2, 3)) &= k + 2 \quad \text{for } k \geq 3. \end{aligned}$$

The collection contains the 61 graphs $P(2k + 1, 2)$ with $2 \leq k \leq 62$ and the 117 graphs $P(m, 3)$ with $9 \leq m \leq 125$.

5.3 Evaluation

We use the following naming convention for the various planarization heuristics: The first part refers to the edge insertion method (*fix*, *var*, *multi*) and the second part to the postprocessing strategy (*none*, *ins*, *all*, *inc*). Later on, we will also introduce a special version for multiple edge insertion with a three-part naming scheme (*none-all*, *inc-all*, *incIns-all*). The following table shows the conventions we use in diagrams for displaying the various heuristics.

⁴Recall that a generalized Petersen graph $P(m, k)$ consists of $2m$ vertices v_0, \dots, v_{m-1} and w_0, \dots, w_{m-1} , paired via edges (v_i, w_i) for all $0 \leq i < m$. The vertex groups are connected via edges $(v_i, v_{(i+1) \bmod m})$ and $(w_i, w_{(i+k) \bmod m})$, for all $0 \leq i < m$.

| Edge Insertion Method (line style) | |
|--|--------------------|
| <i>fix</i> | blue solid lines |
| <i>var</i> | red dotted lines |
| <i>multi</i> | green dashed lines |
| <i>multi</i> with special postprocessing | orange solid lines |

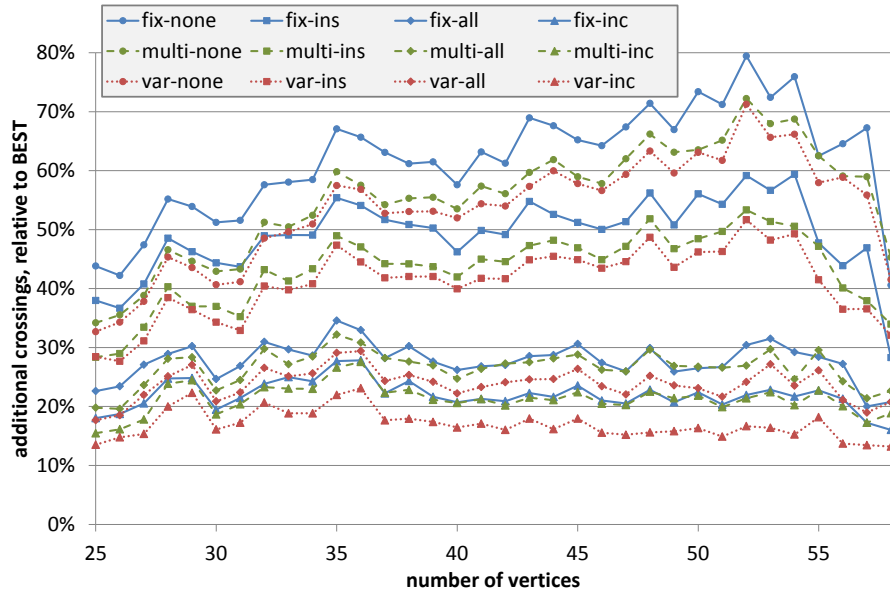
| Postprocessing Strategy (marker for data points) | |
|--|---------------|
| <i>none</i> | ● (circles) |
| <i>ins</i> | ■ (squares) |
| <i>all</i> | ◆ (diamonds) |
| <i>inc</i> | ▲ (triangles) |
| <i>none-all</i> | + (crosses) |
| <i>inc-all</i> | * (stars) |
| <i>incIns-all</i> | – (dashes) |

Solution quality compared to (near) optimal results. For the *Rome* graphs, we know many optimal or near optimal solutions obtained by the exact branch-and-cut algorithm [5]⁵. In the following, we use the term *BEST* to denote the best results we know for the benchmark graphs; these are either from our experiments (including runs with 100 permutations) or from the branch-and-cut algorithm (which gives either an optimum solution, or an upper bound when the algorithm terminates due to its time limit). We analyze the relative difference between heuristic solutions and *BEST*. Since we know the exact solutions for many of the graphs, this gives a very good impression on the actual quality of the heuristics. We note that the exact algorithm is clearly slower than any of the considered heuristics by orders of magnitudes (see [5, 9]), and hence we do not compare runtimes with this algorithm.

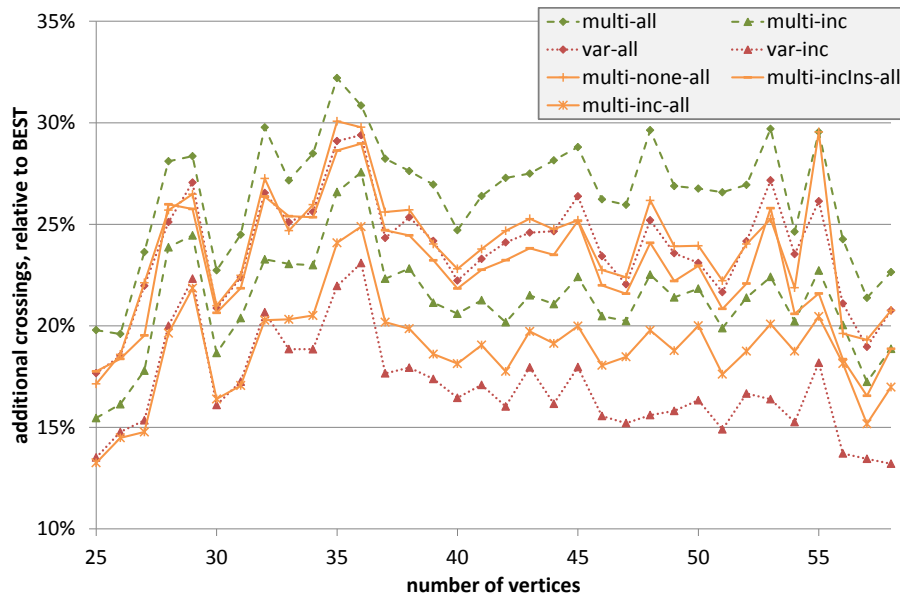
We first study the effect of postprocessing; see Figure 7(a). Our results confirm the findings from [17] that postprocessing helps a lot, and this also holds for multiple edge insertion. Surprisingly, our new incremental postprocessing achieves clearly better results than the previously best *all*, and this holds for all edge insertion strategies, thus supporting the assumption that trying to decrease the number of crossings already for intermediate solutions while inserting the edges one-by-one helps to improve the final solution. We also observe that the advantage of *multi* over *fix* is large without postprocessing, but becomes smaller and smaller the more postprocessing is applied. The main reason for this is that the more postprocessing we do the more the postprocessing phase becomes the crucial factor, and this phase is the same for both edge insertion strategies.

Inspired by this observation, we experimented with an additional postprocessing for the *multi* strategy, where we reused the postprocessing with variable embedding; see Figure 7(b). The variants *multi-none-all* and *multi-inc-all* perform *multi* with no or incremental postprocessing plus postprocessing with

⁵Note that our reduction strategy does not change the crossing numbers for the *Rome* graphs, since they do not contain multiple edges.



(a) basic edge insertion and postprocessing variants



(b) multi edge insertion using postprocessing with variable embedding

Figure 7: Number of crossings for *Rome* graphs, relative to BEST known solutions.

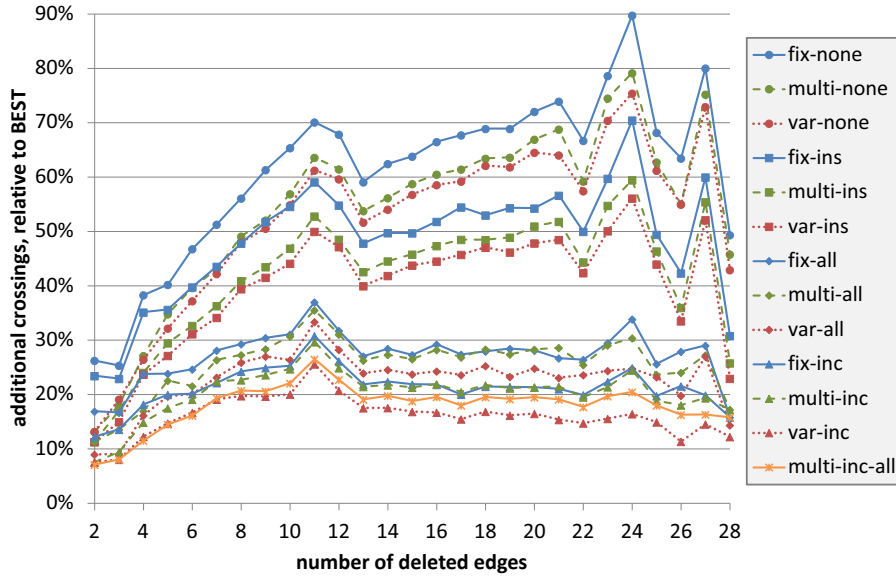


Figure 8: Number of crossings for *Rome* graphs, relative to BEST known solutions, with respect to number of deleted edges.

variable embedding afterwards for all edges; *multi-incIns-all* works similar as *multi-inc-all* but restricts the incremental postprocessing to the inserted edges. Whereas *multi-none-all* and *multi-incIns-all*—which retain the approximation guarantee—are about as good as *var-all*, *multi-inc-all*—which in theory does not give those guarantees—comes close to *var-inc* (for larger graphs, it lies between *var-all* and *var-inc*).

Compared to the (near) optimal results, we can see that the best heuristics are about 15% away from *BEST*. For graphs up to 40 vertices, we know the exact crossing number for most of the graphs (cf. Figure 4), and the results for larger graphs seem to confirm this trend. For graphs with more than 50 vertices, we do not know if *BEST* comes close to the exact crossing numbers, and therefore these results must be treated with caution.

Finally, we wanted to find out if the number of edges to be inserted affects the relative order of heuristics regarding solution quality. In Figure 8 we grouped the graphs by number of edges deleted in the planar subgraph (instead of number of vertices) and show the number of additional crossings compared to *BEST*. We observe that the relative order is the same as when grouping the graphs by number of vertices (see Figure 7) and also remains consistent over the whole range of edges to be inserted.

Running times. We knew already from [17] that the *var* edge insertion strategy is significantly slower than the *fix* strategy, since it requires to compute SPR-trees frequently. It was also known that postprocessing is time-consuming.

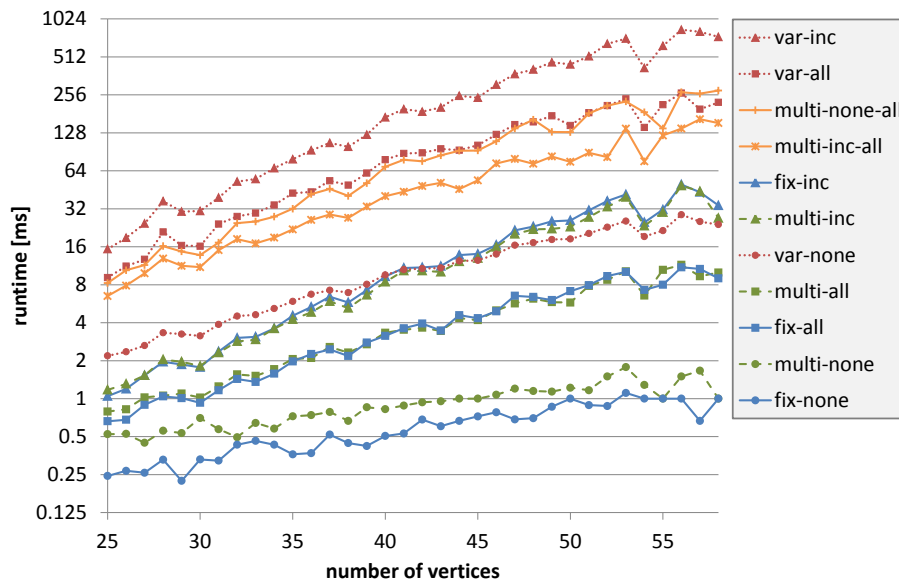


Figure 9: Runtimes in milliseconds for the *Rome* graphs.

Here, we want to focus on the new algorithms: How fast is *multi* edge insertion compared to the *fix* and *var* strategies, and how much slower is *inc* postprocessing compared to *all*.

Figure 9 shows the runtimes on a logarithmic scale. We can see that the overhead of *multi* compared to *fix* is small, and even becomes negligible if postprocessing is used. The *var* variants are always clearly slower, as they require a new SPR-decomposition after each edge insertion, whereas *multi* uses only a single such decomposition. The *inc* variants take about 2–4 times longer than *all*, which is acceptable regarding the achieved improvements in quality.

For our special postprocessing variants for *multi* with additional *var*-postprocessing, we observe that more intensive postprocessing with fixed embedding reduces the effort required with the time-consuming *var*-postprocessing and results in smaller runtimes (i.e., *multi-inc-all* is faster than *multi-none-all*). Since *multi-inc* requires a similar runtime as *var-none* but is in quality even better than *var-all* (the previously quality-wise best known heuristic variant), it is a very good choice in practice.

How much can permutations improve the solutions? Beside postprocessing, we can also use multiple permutations of the order of edges to be inserted for improving the solutions, and permutations and postprocessing can also be combined.

Figure 10 studies the effect of permutations—recall that the prior discussions did not consider multiple permutations. We applied 100 permutations and show the relative reduction of the gap between a single run of the respec-

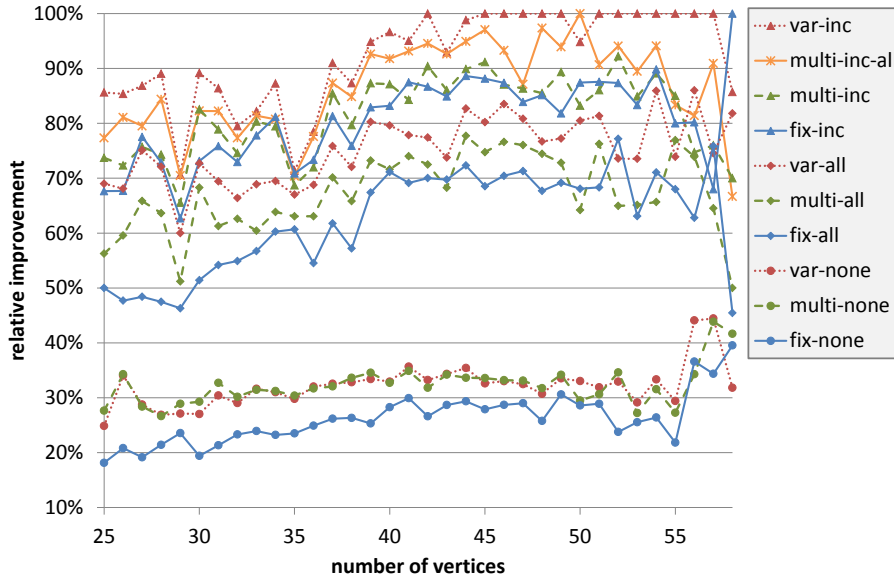


Figure 10: Effect of 100 permutations on the number of crossings (*Rome* graphs).

tive heuristic and *BEST* in the diagram. Hence, 100% reduction means that 100 permutations led to the best solution we know. The main message is that permutations without postprocessing are not very effective, whereas the combination of postprocessing and permutations always gets significant improvements. The incremental postprocessing variant does not only lead to best results (for graphs with 45 or more vertices, it finds the best known solutions in most cases), but is also the most effective one in combination with permutations. In the following, we will append algorithm names with “-100” to denote the variant where 100 permutations are considered.

Solution quality compared to known crossing numbers. The *KnownCR* graphs allow us to further compare the heuristic results with actual crossing numbers. Figure 11 summarizes our findings for some selected heuristics, showing the average relative deviation from the crossing number for the different graph classes. Firstly, the class $P(m, 2)$ of graphs (which all have crossing number 2 or 3) could optimally be solved by all heuristics, hence we omit it in the diagram. For the classes $P(m, 3)$, $G_i \square C_n$, and $G_i \square P_n$, all heuristics perform well, being only 2-13% away from the optimum, and their order with respect to quality is as expected. The class $C_n \square C_m$ shows some unusual behavior: Without permutations, the edge insertion method seems to have only a very small influence on the quality of the solution; with 100 permutations, the *fix* strategy is surprisingly superior to both *multi* and *var*. After analyzing the data, we found that this happens only for a few graphs. In these cases, the distinct runs of *fix* usually find solutions that are a bit worse than *multi* or *var*, but in some

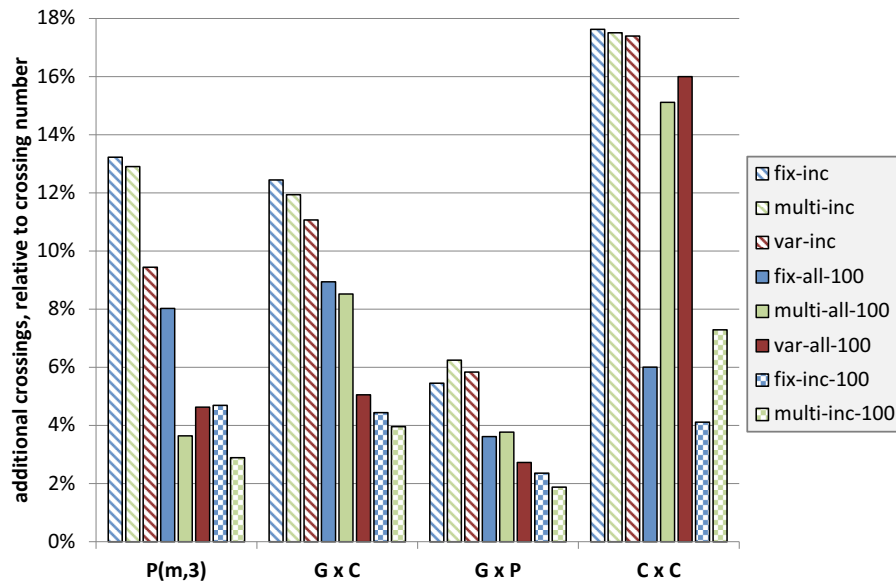


Figure 11: Number of crossings for *KnownCR* graphs, relative to crossing number.

rare cases a much better solution is found. We assume that this could be caused by the fact that accepting worse intermediate solutions while inserting the edges can lead to a better final solution.

AT&T and ISCA graphs. Whereas the *Rome* graphs are fairly homogeneous graphs with a simple structure and the *KnownCR* graphs are artificial, the *AT&T* and *ISCA* graphs are real-world graphs with quite diverse structures.

We first consider the *AT&T* graphs. For analyzing the results, we decided to group the graphs according to the best found solutions (the first group contains graphs with 0, . . . , 24 crossings; the last group graphs with 700, . . . , 799 crossings). Figure 12 shows the relative difference between heuristic and best solution; the horizontal axis shows the 17 groups of graphs described above. We can confirm that incremental postprocessing clearly dominates *all* for all edge insertion strategies, and *var-inc* is by far the best strategy both without and with (*var-inc-100*) permutations. Multiple edge insertion is also slightly better than *fix*. We remark that *multi-incIns-all* (not shown in the diagrams) performs similar as *var-all*, both regarding solution quality and runtime.

However, the domination of *var* comes at a price: Whereas *fix* and *multi* take about the same runtime, *var* is more than 10 times slower (see Figure 13). Hence, *multi-inc* is again a good compromise, as it is even clearly faster than *var-all* (in fact, 3–10 times faster).

For the *ISCA* graphs, we focus on the *multi* and *var* methods. Figure 14 shows again the relative difference between heuristic and best solution, this time for each graph in the benchmark set separately (i.e., the horizontal axis shows

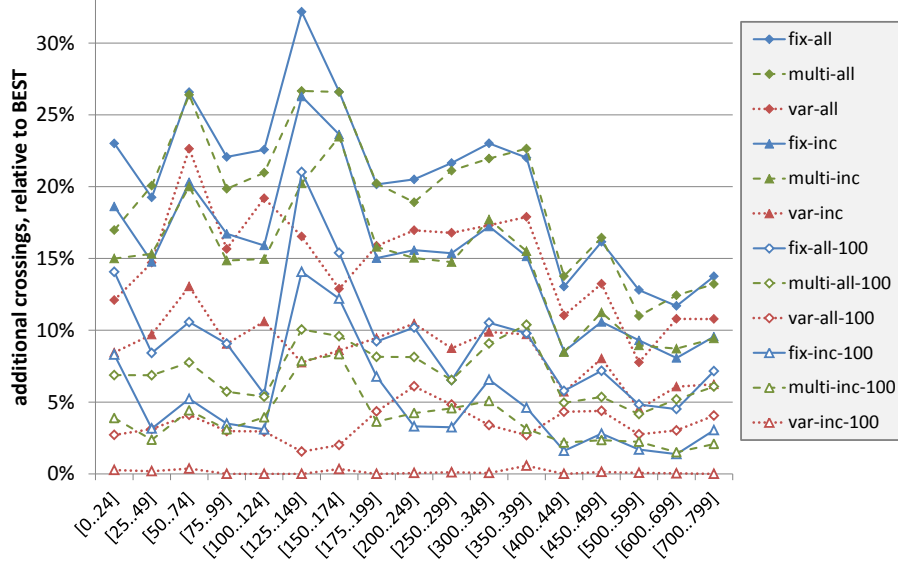


Figure 12: Number of crossings for $AT&T$ graphs, relative to best found solution.

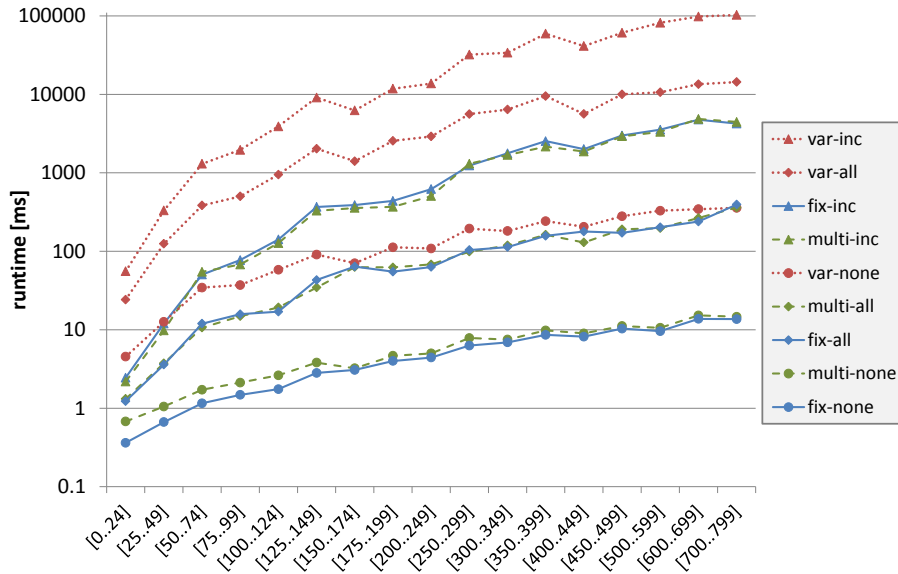


Figure 13: Runtime (in milliseconds) for $AT&T$ graphs, grouped by best number of crossings.

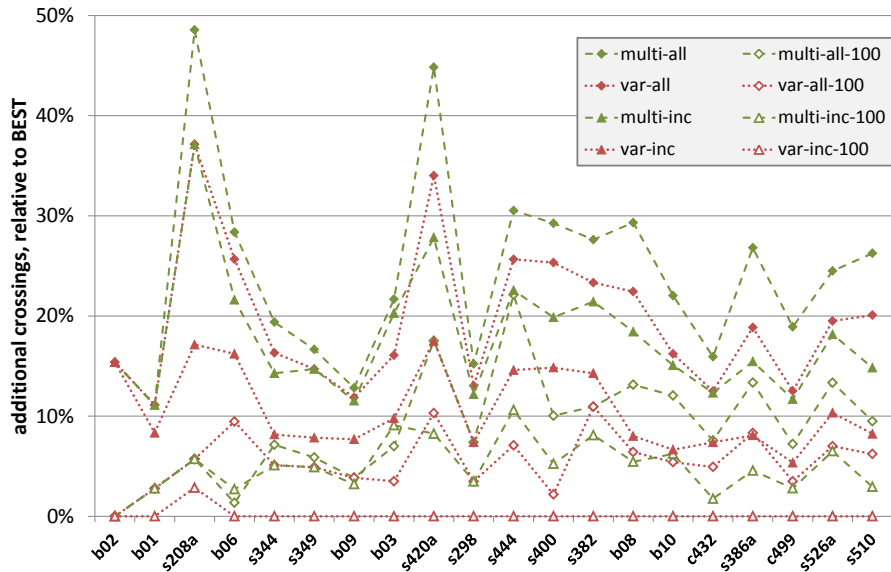


Figure 14: Number of crossings for *ISCA* graphs, relative to best found solution.

the graph instances in the *ISCA* benchmark set). The graphs are sorted by increasing number of edges deleted in the planar subgraph. We omitted the *var* heuristics without postprocessing, namely *var-none* and *var-none-100*, in the diagram; these curves lie between 70% and 100% for the larger graphs, which is a lot more than what the heuristics with postprocessing achieve, underlining again that postprocessing is essential. We can also see that the *multi* variants come quite close to the corresponding *var* variants. This is again accompanied by a much better runtime of *multi* (see Figure 15), in this case *multi-inc* is about 15–30 times faster than *var-inc*, and *multi-all* even about 100 times faster. Also observe that the runtimes for *fix* and *multi* are almost identical.

Multiple edge insertion and crossing number. The *multi* edge insertion strategy computes an embedding of the planar subgraph, based on a voting scheme that considers the optimal edge insertion paths for each edge to be inserted. Thus, this embedding should be a good compromise for inserting all edges. However, when computing this embedding, the crossings that will occur between edges to be inserted are not taken into account, and these can be as many as $\binom{|F|}{2}$, where F denotes the set of edges to be inserted. To conclude this section, we want to analyze if this approach is justified in practice.

Summing up the crossings that would occur if we could independently insert each edge optimally obviously leads to a lower bound for the multiple edge insertion problem (but not for the crossing number of the graph itself). In Figure 16, we consider the *ISCA* graphs and show the additional number of crossings relative to this lower bound; hence the horizontal line at 0% represents

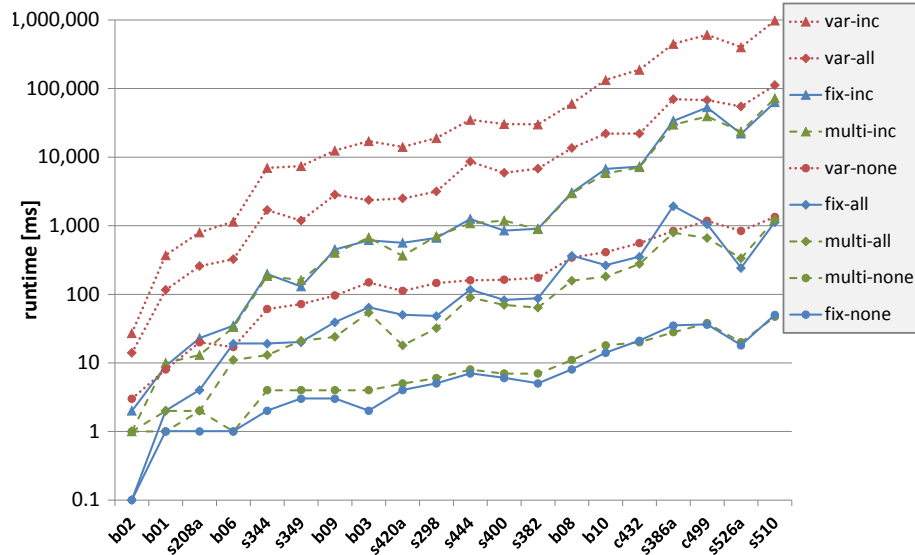


Figure 15: Runtime (in milliseconds) for *ISCA* graphs, sorted by number of deleted edges.

this lower bound. We remark that the number of edges to be inserted is between 9 and 157, lower bounds lie between 12 and 469, and the relative values for $\binom{|E|}{2}$ lie between 300% and 3100%.

The *insertion costs (fixed embedding)* are the number of crossings that occur when we insert the edges into the embedding computed by *multi*, disregarding crossings between the inserted edges. Therefore, this is exactly the value the *multi* strategy tries to minimize. We can see that this value comes quite close to the lower bound in most cases. On the other hand, the curve *best crossings (edge insertion)* gives the best solution we have for the multiple edge insertion problem (hence regarding all crossings). As could be expected, this value is clearly higher (roughly 50%–100%) than *insertion costs (fixed embedding)*, but still low enough such that disregarding the crossings between inserted edges is still meaningful. Finally, the curve *best crossings (all)* shows the best number of crossings we could obtain, and thus also shows what we can gain if we allow that edges in the planar subgraph may cross.

6 Conclusions

We presented *inc*, a new practically dominating postprocessing strategy for the planarization heuristic, and report on *multi*, the first implementation of any crossing minimization approximation algorithm for general graphs. Both algorithms outperform any previously known heuristic in terms of solution quality,

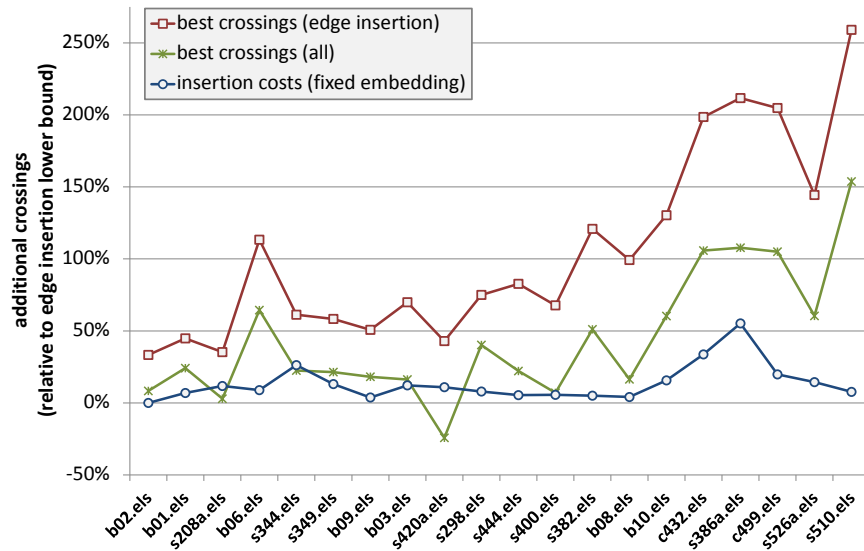


Figure 16: Number of additional crossings for *ISCA* graphs, relative to lower bound for edge insertion costs.

and if one cannot afford the relatively long running times for inserting all edges iteratively into a variable embedding using *inc*, the *multi* variants give the probably best balance between running time and solution quality: while being much faster, its solutions tend to be only slightly weaker than *inc*'s.

The in our opinion most interesting open questions regarding the planarization method are the following two:

Vertex insertion. How does optimal vertex insertion [6] perform in practice, and can the (rather high) theoretical runtime be improved (in theory or in practice)?

Multiple edge insertion. What is the complexity for inserting k edges optimally, where $k \geq 2$ is constant?

References

- [1] J. Adamsson and R. B. Richter. Arrangements, circular arrangements and the crossing number of $C_7 \times C_n$. *Journal of Combinatorial Theory, Series B*, 90:21–39, 2004.
- [2] C. Batini, M. Talamo, and R. Tamassia. Computer aided layout of entity relationship diagrams. *J. Syst. Software*, 4:163–173, 1984.
- [3] L. W. Beineke and R. D. Ringeisen. On the crossing numbers of products of cycles and graphs of order four. *Journal of Graph Theory*, 4:145–155, 1980.
- [4] S. Cabello and B. Mohar. Crossing and weighted crossing number of near planar graphs. In *Proc. GD '08*, volume 5417 of *LNCS*, pages 38–49. Springer, 2008.
- [5] M. Chimani. *Computing Crossing Numbers*. PhD thesis, TU Dortmund, Germany, 2008.
- [6] M. Chimani, C. Gutwenger, P. Mutzel, and C. Wolf. Inserting a vertex into a planar graph. In *Proc. SODA '09*, pages 375–383, 2009.
- [7] M. Chimani and P. Hliněný. A tighter insertion-based approximation of the crossing number. In *Proc. ICALP '11*, volume 6755 of *LNCS*, pages 122–134. Springer, 2011. Full version at ArXiv, id 1104.5039.
- [8] M. Chimani, P. Hliněný, and P. Mutzel. Vertex insertion approximates the crossing number for apex graphs. *Europ. J. Comb.*, 33(3):326–335, 2012.
- [9] M. Chimani, P. Mutzel, and I. Bomze. A new approach to exact crossing minimization. In *Proc. ESA '08*, volume 5193 of *LNCS*, pages 284–296. Springer, 2008.
- [10] J. Chuzhoy, Y. Makarychev, and A. Sidiropoulos. On graph crossing number and edge planarization. In *Proc. SODA '11*, pages 1050–1069. ACM Press, 2011.
- [11] G. Di Battista, A. Garg, G. Liotta, R. Tamassia, E. Tassinari, and F. Vargiu. An experimental comparison of four graph drawing algorithms. *Computational Geometry*, 7(5–6):303–326, 1997.
- [12] G. Di Battista and R. Tamassia. On-line maintenance of triconnected components with SPQR-trees. *Algorithmica*, 15(4):302–318, 1996.
- [13] G. Di Battista and R. Tamassia. On-line planarity testing. *SIAM Journal on Computing*, 25:956–997, 1996.
- [14] G. Exoo, F. Harary, and J. Kabell. The crossing numbers of some generalized Petersen graphs. *Mathematica Scandinavica*, 48:184–188, 1981.

- [15] C. Gutwenger. *Application of SPQR-Trees in the Planarization Approach for Drawing Graphs*. PhD thesis, TU Dortmund, Germany, 2010.
- [16] C. Gutwenger and P. Mutzel. A linear time implementation of SPQR trees. In *Proc. GD '00*, volume 1984 of *LNCS*, pages 77–90. Springer, 2001.
- [17] C. Gutwenger and P. Mutzel. An experimental study of crossing minimization heuristics. In *Proc. GD '03*, volume 2912 of *LNCS*, pages 13–24. Springer, 2004.
- [18] C. Gutwenger, P. Mutzel, and R. Weiskircher. Inserting an edge into a planar graph. *Algorithmica*, 41(4):289–308, 2005.
- [19] P. Hliněný and G. Salazar. On the crossing number of almost planar graphs. In *Proc. GD '05*, volume 4372 of *LNCS*, pages 162–173. Springer, 2006.
- [20] J. E. Hopcroft and R. E. Tarjan. Dividing a graph into triconnected components. *SIAM Journal on Computing*, 2(3):135–158, 1973.
- [21] J. E. Hopcroft and R. E. Tarjan. Efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, 1973.
- [22] M. Jünger, S. Leipert, and P. Mutzel. A note on computing a maximal planar subgraph using PQ-trees. *IEEE Trans. Comp.-Aided Design*, 17(7):609–612, 1998.
- [23] M. Klešč. On the crossing numbers of Cartesian products of stars and paths or cycles. *Mathematica Slovaca*, 41:113–120, 1991.
- [24] M. Klešč. The crossing numbers of certain Cartesian products. *Discussiones Mathematicae Graph Theory*, 15(1):5–10, 1995.
- [25] M. Klešč. The crossing number of $K_{2,3} \times P_n$ and $K_{2,3} \times S_n$. *Tatra Mountains Mathematical Publications*, 9:51–56, 1996.
- [26] M. Klešč. The crossing number of $K_5 \times P_n$. *Tatra Mountains Mathematical Publications*, 18:605–614, 1999.
- [27] M. Klešč. The crossing numbers of products of 5-vertex graphs with paths and cycles. *Discussiones Mathematicae Graph Theory*, 19:59–69, 1999.
- [28] M. Klešč. The crossing numbers of Cartesian products of paths with 5-vertex graphs. *Discrete Mathematics*, 233(1–3):353–359, 2001.
- [29] M. Klešč. Some crossing numbers of products of cycles. *Discussiones Mathematicae Graph Theory*, 25:197–210, 2005.
- [30] M. Klešč and A. Kocúrová. The crossing numbers of products of 5-vertex graphs with cycles. *Discrete Mathematics*, 307(11–12):1395–1403, 2007.
- [31] M. Klešč, R. B. Richter, and I. Stobert. The crossing number of $C_5 \times C_n$. *Journal of Graph Theory*, 22(3):239–243, 1996.

- [32] B. R. Richter and G. Salazar. The crossing number fo $P(N, 3)$. *Graphs and Combinatorics*, 18(2):381–394, 2002.
- [33] R. D. Ringelsen and L. W. Beineke. The crossing number of $C_3 \times C_n$. *Journal of Combinatorial Theory, Series B*, 24(2):134–136, 1978.
- [34] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.
- [35] T. Ziegler. *Crossing Minimization in Automatic Graph Drawing*. PhD thesis, Saarland University, Germany, 2001.

Appendix

This appendix lists the crossings numbers of products of 5-vertex graphs G_i with paths P_n (see Table 1) and with cycles C_n (see Table 2) contained in the *KnownCR* benchmark set and cites the corresponding proofs. Graph $G_1 = P_4$ is not contained in the tables, since products of G_1 with paths or cycles are planar graphs. Similarly, $G_8 = C_5$ is missing in the table for paths as those products are also planar. Moreover, for the graphs $G_{10}, G_{15}, G_{17}, \dots, G_{21}$, the crossing numbers for products with cycles are unknown.

| i | G_i | $cr(G_i \square P_n)$ | i | G_i | $cr(G_i \square P_n)$ |
|-----|-------|-----------------------|-----|-------|-----------------------|
| 2 | | $2(n-1)$ [23] | 13 | | $n-1$ [28] |
| 3 | | $n-1$ [28] | 14 | | $2(n-1)$ [28] |
| 4 | | $n-1$ [28] | 15 | | $3n-1$ [24] |
| 5 | | $n-1$ [28] | 16 | | $3n-1$ [27] |
| 6 | | $2(n-1)$ [28] | 17 | | $2n$ [28] |
| 7 | | $n-1$ [28] | 18 | | $3n-1$ [24] |
| 9 | | $2(n-1)$ [28] | 19 | | $3n-1$ [28] |
| 10 | | $2n$ [25] | 20 | | $4n$ [28] |
| 11 | | $2(n-1)$ [28] | 21 | | $6n$ [26] |
| 12 | | $2(n-1)$ [24] | | | |

Table 1: The crossing numbers of products of 5-vertex graphs with paths used in the *KnownCR* benchmark set.














| i | G_i | $\text{cr}(G_i \square C_n)$ | i | G_i | $\text{cr}(G_i \square C_n)$ |
|-----|---|---|-----|---|--|
| 2 |  | $\begin{cases} 2 & n = 3 \\ 4 & n = 4 \\ 8 & n = 5 \\ 2n & n \geq 6 \end{cases}$ [23] | 9 |  | $2n$ [29] |
| 3 |  | $\begin{cases} 1 & n = 3 \\ 2 & n = 4 \\ 4 & n = 5 \\ 2n & n \geq 6 \end{cases}$ [29] | 11 |  | $\begin{cases} 7 & n = 3 \\ 3n & n \geq 4 \end{cases}$ [30] |
| 4 |  | n [29] | 12 |  | $2n$ [29] |
| 5 |  | n [29] | 13 |  | $\begin{cases} 7 & n = 3 \\ 3n & n \geq 4 \end{cases}$ [29] |
| 6 |  | $\begin{cases} 4 & n = 3 \\ 6 & n = 4 \\ 9 & n = 5 \\ 2n & n \geq 6 \end{cases}$ [29] | 14 |  | $3n$ [29] |
| 7 |  | $\begin{cases} 4 & n = 3 \\ 2n & n \geq 4 \end{cases}$ [29] | 16 |  | $\begin{cases} 3n & n \text{ even} \\ 3n + 1 & n \text{ odd} \end{cases}$ [27] |
| 8 |  | $\begin{cases} 5 & n = 3 \\ 10 & n = 4 \\ 3n & n \geq 5 \end{cases}$ [3, 31, 33] | | | |

Table 2: The crossing numbers of products of 5-vertex graphs with cycles used in the *KnownCR* benchmark set.